Lessons Learned from the Scientist's Expert Assistant Project

Jeremy Jones [b], Chris Burkhardt [a], Mark Fishman [c], Sandy Grosvenor [d], Anuradha Koratkar [a], LaMont Ruley [b], and Karl Wolf [c]

[a]Space Telescope Science Institute, 3700 San Martin Drive, Baltimore, MD 21218
[b]NASA Goddard Space Flight Center, Greenbelt, MD 20771
[c]AppNet Inc., 1100 West Street, Laurel, MD 20707
[d]Booz-Allen Hamilton, 7404 Executive Place, Seabrook, MD 20706

ABSTRACT

During the past two years, the Scientist's Expert Assistant (SEA) team has been prototyping proposal development tools for the Hubble Space Telescope in an effort to demonstrate the role of software in reducing support costs for the Next Generation Space Telescope (NGST). This effort has been a success. The Hubble Space Telescope has adopted two SEA prototype tools, the Exposure Time Calculator and Visual Target Tuner, for operational use. The Space Telescope Science Institute is building a new set of observing tools based on SEA technology. These tools will hopefully be a foundation that is easily adaptable to other observatories including NGST.

The SEA project has aggressively pursued the latest software technologies including Java, distributed computing, XML, Web distribution, and expert systems. Some technology experiments proved to be dead ends, while other technologies were unexpectedly beneficial. We have also worked with other projects to foster collaboration between the various observing tool programs. In two years, we have learned a great deal that will be useful to future software tool efforts. In this presentation, we will discuss the lessons that we've learned during the development and evaluation of the SEA. We will also discuss future directions for the project.

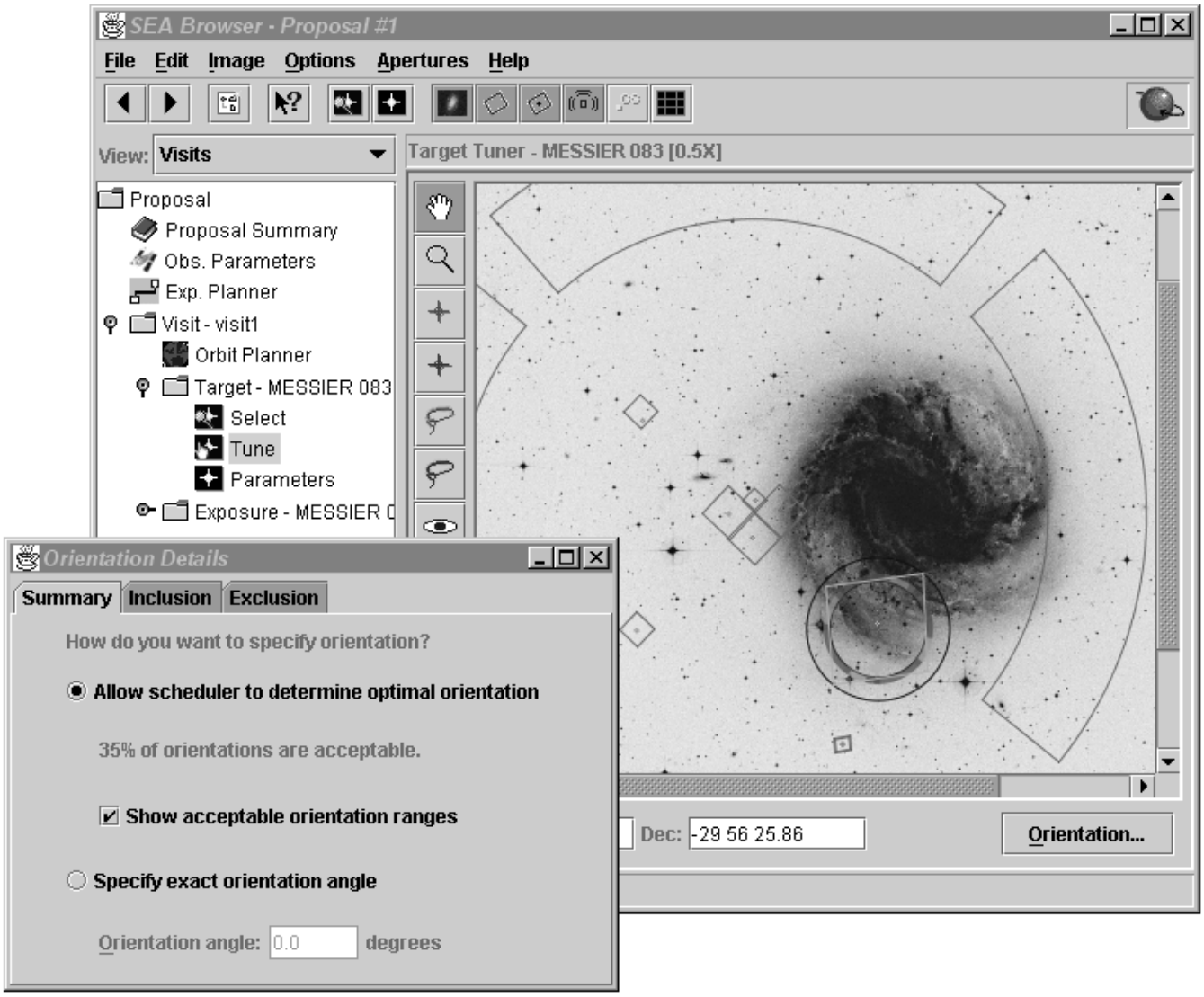
Keywords: Observing tools, NGST, Hubble Space Telescope, SEA, Java, XML, user support, proposal preparation

1. INTRODUCTION

The Scientist's Expert Assistant (SEA) has been a prototype effort to investigate ways to substantially reduce the effort involved in supporting astronomical observing programs. A small team of astronomers and computer scientists for the Space Telescope Science Institute (STScI) and Goddard's Advanced Architectures and Automation (Code 588) group developed a functional prototype that uses an interactive visual and expert system approach to developing and planning observing programs. The system was initially developed with Hubble Space Telescope's Advanced Camera for Surveys (HST/ACS) as a test bed and currently supports several of its instruments.

In SEA, users can retrieve and display previously observed images of astronomical targets, overlay any of HST's instrument apertures on the image. Further, by "clicking and dragging" they can visually define some of the parameters of their observations. For example, the probability of scheduling can be significantly affected by specifying particular aperture orientations. In SEA users can interactively rotate the aperture and see the impact of their orientation constraints (see Figure 1). A built-in Exposure Time Calculator allows users to graphically display the impact of changes in target properties and instrument setups on exposure time and/or signal-to-noise (see Figure 2). In addition, a rudimentary expert assistant can quickly guide new users through the various detector/filter combinations and recommend a combination that fits the users' scientific needs. SEA has also taken the first steps to integrate documentation with software, providing preliminary context-sensitive help, and access to reference data.

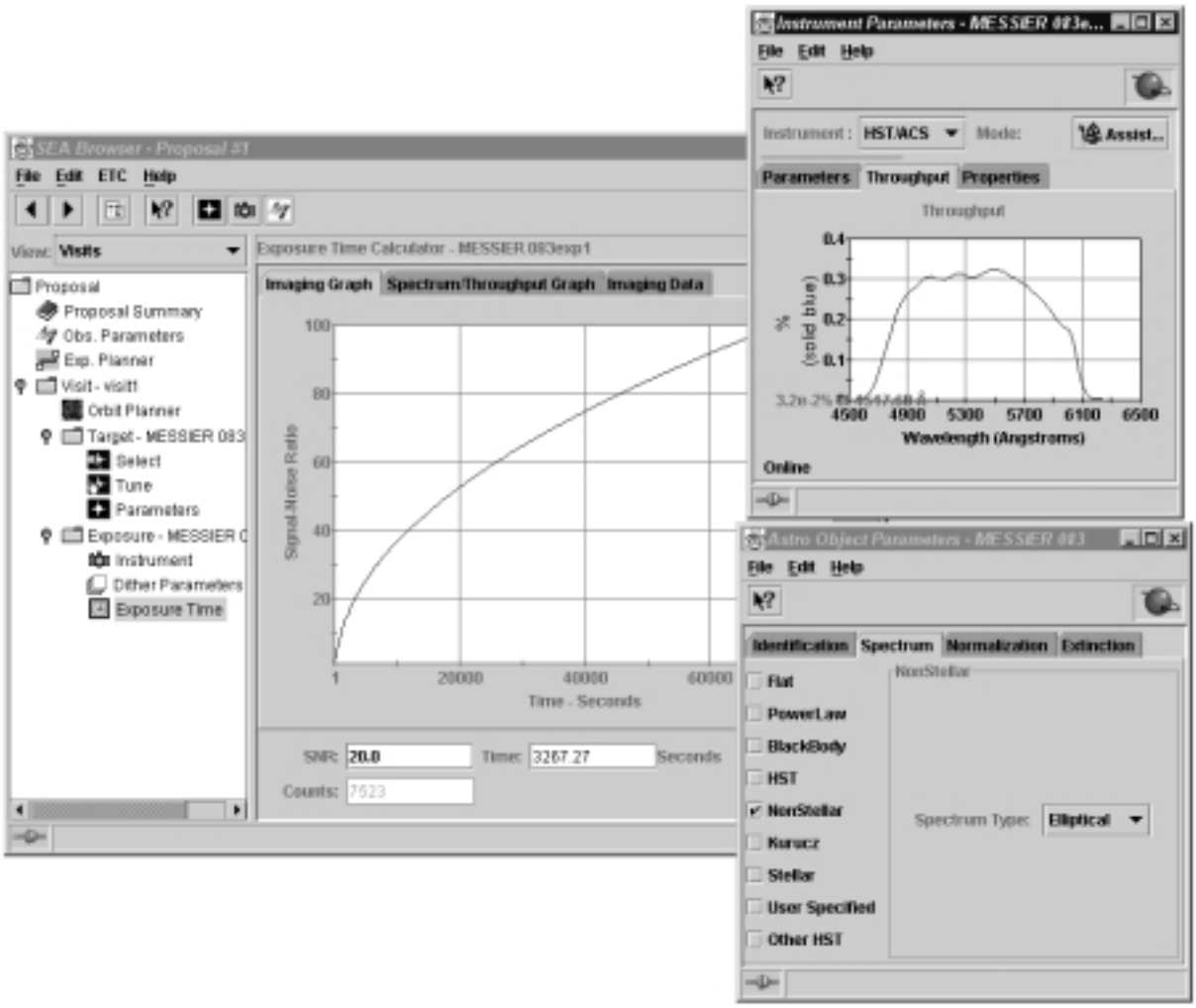
Figure 1 SEA's Visual Target Tuner



SEA is based on a component architecture that shares a common underlying object-oriented design and allows the major components to function either as independent modules, or as an integrated whole under the "Proposal Browser." Current modules include a Visual Target Tuner, an Exposure Time Calculator, a Visit Planner, an Orbit Planner, and the integrating Proposal Browser component. By developing the system using the Java programming language, we have a true multi-platform system that operates on Microsoft Windows, Solaris, Linux and other Unix platforms with no modifications.

The technical approach to developing SEA has been an iterative rapid prototyping approach. Informally known as "design-a-little-build-a-little-test-a-little," this approach involves iterating through the design/build/test cycle allowing new ideas to be quickly explored, and if promising to be developed further. If a particular feature does not show effectiveness, it can be abandoned before excessive resources have been invested.

Figure 2 SEA's Exposure Time Calculator



2. USER INVOLVEMENT AND TEAM INTERACTION

SEA's approach to incorporating user input has been an iterative approach that has incorporated several different components. As one of the first steps to the project, we conducted interviews with several astronomers, and STScI staff about their perceptions of the strengths and weaknesses of their current tools. This included some initial brainstorming about what types of new tools might have the most significant impact. Throughout the project life, we have tried to demonstrate SEA at major astronomical conferences. The feedback we've obtained through these conferences has had a significant impact on the project. Lastly, we recently ran the SEA through a more formal evaluation phase.

Perhaps the most critical component of incorporating user involvement has been the addition of a practicing astronomer as a full member of the team. We have coined a new term in computer usability for such a team member: "alpha-user." While most of the team is composed of computer scientists with little formal astronomical training, our "alpha-user" has several years of experience at STScI in user-support with little formal computer training. An alpha-user acts not only as a primary direct source of end-user input, but also coordinates input from many other users. While primarily acting as an advocate of end-users and sharing the astronomical expertise of our user community, an alpha-user also knows (or learns) enough of the

technical design language to provide an invaluable link between the technical computer developers and the end-user community. SEA has benefited hugely from this experience. The team needed a strong end-user with expertise in astronomy as well as a vision of the types of tools needed.

During February 2000, after two years of informal demonstrations, SEA underwent a more formal evaluation period. We established a list of specific tasks for our evaluation team of working astronomers to perform. We then observed them as they performed those tasks providing minimum coaching necessary. Details on the results of this evaluation phase can be found in the poster/paper by A. Koratkar, et al[2]. This phase has been extremely beneficial. Having a set of novice SEA users perform a set of specific tasks added a whole new level of feedback. We flushed out a number of undiscovered bugs, got some excellent ideas, and learned a tremendous amount about what was and was not intuitive. *We strongly recommend incorporating formal end-user evaluations throughout the development and testing cycle of a new product. Don't wait until the end of the development cycle. Use formal evaluations early and often. It will save valuable time and effort.*

An often-asked question about software development is: Can "rapid prototyping" really work? For SEA the answer has been a resounding yes. Our objective during SEA's development has been to test and evaluate new visual concepts, not generate a production system. However, we also knew at the beginning that even as a prototype, the software engineering needed to be sound, or it would not support the infrastructure that we knew SEA had to support. One of the significant aspects of SEA has been a very loose coupling of requirements to system features. Since we were intentionally exploring new interface concepts, we had very few specific requirements. The rapid prototyping allowed us to adjust to an evolving and ever-changing set of requirements and priorities. That coupled with the object-oriented approach has also allowed the underlying architecture of SEA to evolve and grow with the system.

3. TECHNOLOGY

3.1. Java

At the inception of SEA, Java was just beginning to emerge as a truly viable development language. Its claims of cross platform independence, its extensive libraries, and its ability to run from the web all looked like a good fit for meeting SEA's goals. Therefore, the decision was made to design and implement SEA as a pure Java product. Since this decision there have been many issues and trials but overall the feeling of the team has been that this decision was a correct one.

One of the early-recognized strengths of Java was the availability of many built-in and add-on libraries. Libraries such as Java Advanced Imaging (JAI) and Java Project X (XML) as well as the more standard Swing and Java2D libraries allowed us to focus our effort on developing higher level, product specific components. These built-in libraries made for a substantially smaller development time than might have been expected given a less robust set of development tools and it is our belief that SEA would not be nearly as rich a tool without the advantages that these libraries provided.

On the other side of our decision to use Java, one of the early areas of concern for SEA was performance. Java, with its use of byte code and a virtual machine (VM), has frequently been blamed for performance problems. Performance has at times been an issue for SEA. We adapted partly through some design modifications, but more often Sun resolved the issues in subsequent Java releases. The optimizations we made included everything from redesigning algorithms to just placing large tasks on a separate thread to allow the user to continue while the task completed. Probably one of our most notable performance problems dealt with the startup of the application. The starting of the VM, the loading of all jars, and the initialization of the SEA application could frequently take 20 seconds or more. To help minimize this cost we threaded what we could of this startup and made sure that a splash screen came up at the earliest moment possible. This solution, while not perfect, seems to be acceptable. However, there is hope for the future. Based on documentation from Sun as well as some preliminary, fairly unscientific performance tests the Java 1.3 client VM significantly improves Java's performance. For example, the startup of SEA went from 20 seconds to 11 seconds. Other areas of SEA also received performance improvements from the new VM.

Another issue that was less obvious but quickly became a concern for us was Java's rapid life cycle. The SEA project stretched from the beginnings of Java 1.1 until the current candidate release for Java 1.3. During this time Java went through several evolutions. Bugs were fixed. New bugs were created. Libraries changed. APIs changed. Even class behaviors changed in certain cases. However, surprisingly, the majority of these changes required relatively low amounts of rework in the SEA code. There were times where features needed to be disabled until a bug fix came out or occasionally method

parameter lists changed but in general it was our experience that Sun went out of its way to make the impacts of their changes as small as possible.

One important area for the SEA team dealt with Java's cross platform abilities. The "write-once-run anywhere" claim was a key feature in choosing Java as our development language. We needed to support customers who worked in Unix, NT, and Apple environments. While there have been issues relating to this cross platform capability, some minor problems with fonts and layouts, all core functionality ended up porting fairly easily. Further, many of the problems with cross-platform consistency have been eliminated by Sun as Java has evolved. We developed SEA on NT but were able to deploy it on Solaris, Linux, and even DEC Alpha machines. At the time of this paper, Apple still has not released a version of Java 1.2 for the Macintosh, so SEA was unable to run in that environment but other than that we were pleased with Java's support of this capability.

One of our early goals called for SEA to run as both an application and an applet. While not absolutely necessary this requirement appeared to be fairly easy to implement and was useful in allowing users to always run the latest version of the application. However, we concluded that applets are best for simple applications with no major security requirements. The reason for this decision dealt with three problems that arose during development. The first one was program size. Between the SEA classes and third party libraries, we found that our distribution file ballooned to a multi-megabyte size (approximately 8 megs). This is substantially beyond the size threshold for reasonably running as an applet.

The second problem dealt with the Java security model. In JDK 1.1 the security model was coarse and each browser vendor was expected to implement their own version of it. With the coming of the plug-in and JDK 1.2, Sun provided a more fine-grained security model that could be treated the same across all browsers. Unfortunately, while the model itself was better, it required the end-users to partially manage the security setup on their own system. We felt that these end-user setup requirements were too complicated for the average user. While ideas such as developing an installer to setup the applet security were discussed, the decision was eventually made to just drop the applet support entirely.

The third problem with applets is the ongoing lag in major browsers supporting new versions of Java, and of incompatibilities between different browsers. While using Sun's Java Plug-in helped considerably, it required the end-user to install the plug-in, which eliminates much of the advantage of an applet's "transparency".

In dropping support for applets, we still needed a flexible distribution solution that was as simple as possible for our end-users. Eventually we decided on a two-prong method for distributing SEA and keeping it up to date. First, we decided to use InstallAnywhere by ZeroG Software to distribute SEA. InstallAnywhere is a third party tool that allowed us to create installers to distribute SEA, the Java runtime, and any third party libraries on both NT and Unix environments. Further, to make sure that SEA was up to date, we added the ability for it to check its version against the newest version of SEA on the SEA distribution site and download a copy if a newer version existed.

One of the major lessons that we learned while developing SEA was that by taking advantage of the built-in features of Java it is possible to come up with a very plug-able/extensible type of infrastructure. By using common object-oriented principles like interfaces and base objects in the SEA design, the SEA allows the extension of these objects without the need to modify the classes that use those objects. Additionally, by using common software design "patterns" coupled with Java features like dynamic class loading and introspection we have developed an architecture where features can be changed and new components can be added to SEA without the need to modify existing components. One of our better decisions was to model SEA after the Model/View/Controller (MVC) design pattern, which describes a way to separate the system's data from the various user interfaces to the data, while ensuring that changes to data are propagated properly throughout the application. While this pattern is not unique to Java, it is one that the Sun developers emphasize and we found that it made integration much easier. By using one model, which many views shared, we were able to have any changes made by one view seamlessly reflected in all of the others.

3.2. Expert Systems

Expert System Technology provides a way to incorporate expert knowledge of a particular domain into a software system. During the development of the SEA, we attempted to incorporate expert system technology in three different ways. Each technique met had its varying degrees of success which are discussed briefly here. A more complete discussion of these

efforts can be found in paper **4010-25,** "Expert System Technology in Observing Tools", by K. Wolf, et al[1], found elsewhere in these proceedings.

In an expert system, the domain knowledge is coded as a set of rules and data forming a rulebase. You can think of it this way, rather than hard code the domain rules in a programming language such as Java, the rules are supposedly nonprogrammer readable statements stored in a rulebase. The Expert System vendors stress that rules are easier to read than regular programming code and can therefore be verified and maintained by the domain expert. We consider this idealistic. Rulebases are quite difficult to build correctly.

The first task in rulebase development is to gather the "expert knowledge". The process is similar to requirements gathering in a software engineering effort. Trying to extract domain specific rules from the experts is often not an easy task. Some experts have difficulty expressing their "rules". For a variety of reasons, some experts may be reluctant to share their wisdom. Just finding the right questions to ask may require more than a minimal understanding of the subject domain. The rulebase development process usually becomes an iterative exercise.

Many times rules derived from a variety of sources can be contradictory. This often happens when discussing the domain with more than one expert. Each expert may have their own "rules-of-thumb" which may conflict with other expert's rules or even their own. These contradictions must be resolved, otherwise the rulebase will likely provide conflicting results. One potential scheme might be to apply a weighting or priority system to rules. Some rules could be deemed more important than other rules. The expert system we used (Blaze Software's Elements Advisor) provides a priority scheme where some rules can have priority over other rules. However, we did not make use of this scheme. How would you determine which rules should have priority? The better way to resolve the issues is to discuss the problems with the experts and let them resolve the problems.

Once we have the handwritten collection of rules from the various experts, we then have to translate them into expert system rules. The rules may have been expressed as thoughts and ideas, not necessarily in straightforward IF-THEN-ELSE statements. As with other software endeavors, the translation process is more of an art rather than an algorithmic process. Interpreting a rule-of-thumb into one or more rules can be quite difficult. This is where the expert system programmer applies his/her magic.

During the early development of the SEA, the underlying expert system software required the rulebase to be in one large monolithic file. Complex rulebases were difficult to create and control. Later versions of the expert system software have had a more modular approach to the rulebase. Related rules could be grouped together in "rulesets" which could be applied as needed. We strongly recommend you take advantage of rulesets. They dramatically reduce the overall complexity of the rulebase by eliminating the need for state flags and extraneous rules to control rule flow.

When is the knowledge base complete? How is the rulebase verified for accuracy? These are difficult questions to answer. Simply showing a rulebase listing to the experts is usually met with less than enthusiasm and is often counterproductive. The experts are generally not programmers and cannot be expected to be able to read and understand a rulebase listing. This is not to say the experts lack some mental capacity that programmers have, rather, rulebase reading is not their area of expertise.

One solution to the verification process is to use "scenarios". Much like software system testing, several functional scenarios can be devised for the rulebase application. The scenarios are devised in collaboration with the experts. The experts should understand the scenarios and they should be able determine what results to expect, and the rulebase should produce results similar to the expert's results. It is not unusual for the expert system may have different or surprising results than what the expert predicted. The cause might be the expert may apply his own rules in an inconsistent way, whereas the expert system always applies rules in a consistent manner. These differing results must be accounted for. A determination must be made between the rulebase producing surprising results versus the rulebase producing incorrect results.

Once the rulebase has been established, a determination about how the rulebase is best utilized in the main application must be made. A few points should be considered:

- Our experience is with an expert system implemented in Java. In order to use the expert system in the application, its classes must be loaded, run, and the rulebase must be loaded and interpreted. All this takes time. You want to avoid long delays when dealing with impatient users. The suggested technique in Java, is to load and run the expert system

in a separate Thread from the main Thread. The Thread priority should be set to a reasonable level so as not to delay the rest of the application and to ensure a prompt interactive environment to the user.

- One of our first attempts at integrating an expert system into the SEA involved an interrogation of the user through a set of rulebase-generated questions. You have to be careful not to ask too many questions or the experience may be annoying to the user. In particular, if the user happens to be an expert or experienced user, the questions may get in the way of the real work the user is focused on doing. The mechanism must adapt to the capability of the user.
- We strongly suggest the extensive use of rulesets in your rulebase implementation. The benefits of modularization greatly improve the final rulebase.

Expert system technology is not new, but it is new to many programmers. We see a lot of potential for the technology when applied to applications such as the SEA. Anywhere you have users who are both domain experts and novices, you may have a candidate use for an expert system. Rulebase writing is not harder to write than straight compiled code. There is certainly a learning curve, but no more so than learning a new programming language.

We have had mixed success integrating the expert system technology into the SEA. As we have become more experienced, we have learned better techniques. We have plans for an expanded role of the expert system within the SEA (see paper by K. Wolf, et al[1] for more details).

3.3. User Interface Challenges

The SEA design required a display that represented the contents of the proposal to the user. This user interface (UI) would display the high-level proposal elements along with operations to perform on those elements. This UI would form the foundation for navigation between the various tools. From a computer scientist's view, the obvious answer to this problem was a hierarchical view. After all, an HST proposal is composed of a number of Visits, each of which has a Target and one or more Exposures. So the original tree navigation view was born. We quickly realized, however, that this was not the most user-friendly presentation. Indeed, users often have difficulty with the tree view. One reason for this is that the tools are hidden within the tree, which means that the user must drill down into the tree to discover the tools.

Despite the fact that the tree is an imperfect design, we have yet to find a superior design. One refinement of the tree design, however, has at least been a minor improvement. We discovered that users approach the problem of observation definition from different directions. Some users work from a top-down approach, defining their visits, then exposures, while others wish to define their exposures then assign them to visits. Other users prefer to define their targets first. To accommodate these different views, we added the ability to reorder the tree into different "views" on the proposal. The user has the ability to quickly reorder the tree to highlight Visits, Exposures, Targets, or Instruments.

Another interface challenge has been how to handle the different platform assumptions of individual users. Since the SEA is a platform-independent application, a decision had to be made regarding which platform features to emulate. Our goal was to make the SEA look and feel like a modern Microsoft Windows application. Windows was chosen instead of Unix/Motif simply because Windows contains many more useful user interface concepts (even if they did not originate in Windows). The idea was that if the SEA was similar to other tools with which the user was already familiar, the initial learning curve would be less. However, we were concerned that users who were not familiar with Windows (still a sizeable portion of the scientific community) would have difficulty with some of the standard Windows concepts. These concerns were realized in the evaluation. The ability of a user to intuitively grasp features like Tool-Tips, drag and drop, and multi-selection, seemed to correlate with the user's Windows (or Macintosh) experience. When designing a platform-independent application, one must consider that many users may not be familiar with the "standard" features of another platform's user interface.

The original user interface design for the SEA was very different from the current design. An original design goal was to accommodate two categories of users, novice users and experienced users, through two different high-level user interfaces. The browser UI would provide flexible access to every element of the proposal and was intended for the experienced user. The novice UI was designed to provide a simplified means by which the software would walk the user through a series of questions that filled in the details of their proposal. The challenges that we faced in developing this "Interview" user interface are described in a separate paper by K. Wolf, et al[1].

3.4. XML

The SEA uses XML for proposal storage and for configuration and preference files. Originally, we stored proposal information by implementing the Java Serializable interface. Unfortunately, implementation on our rapidly changing prototype fell behind resulting in data files that could not be reloaded.

The solution to our proposal serialization problem came from the Koala Object Markup Language project (http://www.inria.fr/koala/XML/serialization/) and their XML serialization tools. Koala's serialization API uses reflection in addition to the Serializable interface to serialize the graph of objects that form an SEA proposal. It then reforms the resulting data stream into XML. Furthermore, Koala provides an API that uses the SAX parser to reconstruct the graph. All this is performed without having to write explicit serialization methods. While we are able to use the API to read and write proposals, the Koala serialization does not result in very human readable code when applied to our data model. A more human readable XML format that would allow users to read and update proposals outside of the SEA should be considered in the future.

Storing user-settable preferences was a different matter. We initially used our own text-based format for preference storage that resembled C code. While readable by software developers, we decided a more user-friendly format would be desirable. Editing the original "JavaCC" files by hand was also tedious and error prone. Switching to XML allowed us to take advantage of the multitude of XML editors and parsers, and reduced our level of maintenance effort while increasing the readability of our preference files.

3.5. Interfaces to Legacy Systems

Since many institutions already have collections of programs for providing technical information, for example target visibility, optimal roll angle, guide star availability, etc., for the HST observatory, we decided interfacing to legacy systems would be a worthwhile experiment. Instead of rewriting code for target visibility, we decided to interface to the Space Telescope Science Institute's Spike scheduler.

Unfortunately, Spike is a Lisp program running on Solaris. Hence, instead of porting Spike to MS Windows we decided to use Spike's socket interface that gave us the ability to directly connect to a Spike process and query it for data. Further, instead of writing socket calls to Spike directly from the SEA application, we wrapped Spike in a package outside of the SEA called JSpike. This provides the SEA with a generic interface for accessing Target Visibility Data whether online or from data files without needing explicit knowledge of the scheduler.

Further abstraction will allow us to access other legacy systems that implement our scheduler interface. By separating the legacy components into individual jar files we can provide easy substitution of legacy system components by simply replacing jar files.

4. EVALUATION

During February 2000, we conducted user evaluations on SEA. Our evaluators were chosen from astronomers with accepted HST Cycle 9 programs that focused on HST's WFPC2 and STIS instruments. In this paper, we'll briefly summarize the process and what we learned from it. For greater details on the evaluation process and ratings, see paper 4010-27, "Usability Testing and Evaluation Plans for the Scientist's Expert Assistant", by A. Koratkar, et al[2], in these proceedings.

We originally planned two evaluation phases: first, our evaluators would run some test scenarios that would familiarize them with SEA's features; second, the evaluators would use SEA to develop their phase II Cycle 9 proposal as much as they could given SEA's current feature set. We quickly learned that our user community was resistant to committing a large block of time upfront for the evaluation. We adjusted our plan to reduce the planned time to 2 hours (although some evaluators actually spent up to 5 hours). To maximize the feedback with only a 2-hour evaluation window, we tried to accommodate each evaluator's scientific interests as we introduced them to the SEA. We focused on ensuring all evaluators were introduced to key functionalities that they were to evaluate, but had to eliminate the second phase of the evaluation (attempted generation of a full Phase II proposal).

During each evaluation, the evaluator did all the computer input, and was observed and coached as needed by a team that consisted of the SEA evaluation coordinator, an HST support staff observer, and an SEA developer. The coordinator guided the evaluator through a pre-defined set of tasks designed to span SEA's capabilities. The coordinator provided coaching as needed but tried to allow the evaluator time to discover the solution on their own. The support staff observer and developer took notes on the results, process and reactions of both the evaluator and SEA.

After completing the online tasks, our evaluators completed a survey the asked them to rank several features of SEA in both quality and value.

Highlights of what we learned:

- We waited until near the end of the NGST-funded portion of SEA to do a formal evaluation because we felt it was important to wait to evaluate a more complete system, rather than test half-implemented ideas. However, we underestimated the tremendous value to the developers in seeing outside users test the system. We feel now that we should have small formal evaluations throughout the project. We also feel strongly that the developers must be part of the team observing the evaluation. The developers learned more by seeing it happen rather than waiting for a second-hand report.
- We anticipated that our better-developed, more mature modules would get a better reception, but we were surprised at the high correlation between the maturity of a module, and how important the evaluators ranked the functionality.
- We found that users perceive certain concepts very differently. For example, users appreciated and liked the context sensitive help and access to reference information, but when we asked them the question "Would you like documentation to be integrated with software" they ranked it as a very low priority.
- By design, SEA does not force a user to develop an observing program in any particular order. Some users quickly adapted and valued this flexibility; others were frustrated by the apparent lack of focus and direction provided by the tool.
- Our attempts at providing expert help were appreciated, but it was felt that more work needed to be done before these ideas were ready for operational use, again indicating the ranking depended on the maturity of the module.
- The instantaneous updating of information was seen as a two edged sword – there were occasions when this was considered essential (i.e. in the VTT), while in other instances it was found to be unacceptable (i.e. in the ETC where users often want to change several parameters at a time and then see the impact).

Our evaluators ranked the following capabilities very highly: manipulating the telescope's field-of-view, connecting to archives, accessing up-to-date reference information, ability to explore the proposal parameter space. Overall, the SEA was given an average rating of 1.7 on a scale of 1 (excellent) to 5 (poor).

5. COLLABORATION

At the 1998 SPIE meeting on "Observatory Operations to Optimize Scientific Return" there was consensus that we explore the mutual concerns and constraints of the various observatories, so that we can understand where joint, shared, or collaborative effort might be beneficial. We therefore organized the "Workshop on Observing Tools" in October 1998. Over fifty representatives, a mixture of astronomers and software developers, representing a variety of observatories (ground and space-based, optical, x-ray, infrared) attended this workshop. Our goal in organizing this workshop was to promote collaboration between observing tool development teams. It was our hope that this meeting would begin a dialogue on collaboration and software reuse that would continue after the workshop was concluded.

The meeting went quite well. It was well attended, with representatives from many different organizations including Chandra, ESO, Gemini, HST, NOAO, SAO, and IPAC. The emphasis on discussion over presentation was clearly the correct choice. Topics ranged from the commonalities between observatory needs, to the future of observing tools, to how to break down barriers to collaboration. More information on the results of the meeting can be found at http://aaaprod.gsfc.nasa.gov/workshop/.

While the meeting was declared a success, the group hoped that the meeting would be just the beginning of a larger dialogue and collaboration effort. Six working groups were setup that covered specific observing tool topics: Target Visualization Tools, Exposure Time Calculators, Defining an Observation, Optimizing Calibration Observations for Ground Based Observatories, Common Observatory Definition, and Data Services. Both short-term and long-term goals were established for each group. However, despite the best efforts of the members, none of the working groups ever produced any substantial

results. Some groups started out strong but quickly fizzled out due to lack of interest or time, while others never really started at all.

While it is certainly worthwhile to strive for collaboration between organizations, it is unfortunately extremely difficult to sustain. Clearly, the workshop attendees wanted to share their efforts and work towards common solutions. But we were hesitant to push for firm pre-established “deadlines” for the groups to report back their progress, and we still failed to keep the momentum moving. Lack of organizational commitment is one reason. Another is the simple reality of daily pressures exerting priority over other "optional" activities. For a collaboration effort to succeed, it must be carefully focused so that the long-term cost of not participating is higher than the initial cost of participating in the effort. We also believe that distance discourages collaboration, if groups can make the time to get together face-to-face, much more can be accomplished in a shorter time-frame.

The JSky project is one example of a successful collaborative effort. JSky is ESO's attempt at coordinating a library of reusable Java components for use in astronomy. The Visual Target Tuner (VTT) module of the SEA now uses JSky components for its underlying image rendering engine. This includes a FITS image “codec” for the Java Advanced Imaging (JAI) API. JSky enabled the VTT to utilize the JAI libraries with minimal effort, increasing the capability and performance of the VTT. The SEA team has also shared code and ideas that have been incorporated back into JSky, such as a world coordinate system library that was originally written for the VTT. JSky has been a successful collaboration for the SEA because it has actually reduced development time instead of increasing it. Still, JSky is a rather modest library, and we hope that the community will embrace it by contributing additional components.

6. FUTURE DEVELOPMENT

6.1. Simulation

Now that the basic SEA tools and infrastructure are available, we can consider prototyping more advanced tools that are built on that infrastructure. One such idea is the concept of observation simulation. We hope to add to the SEA framework additional tools that will allow astronomers to explore the target/instrument/observatory parameters and then “simulate” the quality of data they will attain. For example, a bright target within (or slightly outside) a field of view can have undesirable effects when observing faint sources. These effects not only depend on the location and brightness of the target but on things such as spectral energy distribution (SED), choice of filters, and exposure times. The simulation will convolve the target properties with the instrument setup, known sources of detector defects, quality of available calibrations, and other observing constraints for a given observing mode. Once the basic model is generated, the user will have the ability to interactively manipulate the various observing parameters to visually determine the impact.

For displaying the simulation, we hope to research visual technologies that are just reaching the average desktop such as real-time 3-dimensional rendering. We plan to evaluate the effectiveness of different visual and interactive approaches, and then focus on further developing those that have the biggest impact to our unique astronomical end-user community.

6.2. Natural Language Interface

The Interview user interface prototypes in SEA were built on the assumption that the user is required to describe in great detail every aspect of their proposal. The interview attempts to offer assistance with a few of those details (filter and detector selection, for example), but the real power of this interface is lost because the system requires detail that the interview format cannot provide easily. These details are required, for the most part, so that the observer may optimize their program. However, if we imagine for a moment that the detailed specification of the proposal by the observer is not necessary, we can envision an interview user interface that allows the user to express their proposal in the simplest form possible. Perhaps this UI would mix natural language processing with the question and answer format. For the simplest programs, only a few questions would be necessary. For more complex programs, the tool would know to ask additional questions when more information is required. Other tools might be integrated to confirm steps during the course of the interview (e.g. the VTT could be used to quickly confirm the target field).

This relatively simple proposal would then be fed into the back-end system, as it is now, but that system would include an optimization step in addition to the scheduling and validation steps that now exist. If it worked, this approach would greatly ease the user's proposal definition requirements, while possibly increasing efficiency for the observatory. Increased efficiency

might be possible because optimization could occur at the observatory level with exposure granularity, rather than per proposal with visit granularity.

6.3. Adaptation to Production Environments

We are continuing to work closely with STScI to adapt at least two of the SEA's tools, the Visual Target Tuner and the Exposure Time Calculator, to a production release for HST during 2000. This entails reviewing the underlying architecture to ensure that it has robustness for a production rather than prototype environment, reviewing the scientific algorithms closely to ensure accuracy, extending the end-user documentation substantially, and establishing a successful strategy for sharing code between multiple developments teams in disparate locations (STScI and Goddard) during parallel development. This effort lays the groundwork for SEA to grow into a multi-observatory tool. We have also begun to discuss possible collaborations with other observatories including Gemini, SIRTF, and SOFIA.

7. CONCLUSION

At the start of the SEA project, NGST funded a research effort to evaluate ways to dramatically reduce the time and effort required to support a general observer program. We have looked at a number of possibilities, some like the Visual Target Tuner show great promise, while others such as expert system applications remain intriguing but are still not successful. We strongly believe that visual technologies are a substantial improvement over previous tools, and their impact may well be in areas that are different than we initially expected. We think the major cost savings will come in the eventual merging of Phase I and Phase II tools, along with support for exploration of an astronomer's initial concept development and research. Additionally, we believe that the trend, while not yet complete, towards collaboration and an evolution to multi-observatory tools is the avenue that promises the greatest cost benefit. Consider the cost benefits to observatories if they can adapt a commonly used tool, rather than redevelop (and subsequently maintain) a custom tool. Consider the productivity impact to both support staff and astronomers if astronomers can use the same basic software suite to develop proposals end-to-end for different observatories.

8. ACKNOWLEDGEMENTS

This work was supported by the Next Generation Space Telescope (NGST) technology funding. The SEA is a joint effort between the Space Telescope Science Institute (STScI) and the Advanced Architectures and Automation Branch of Goddard Space Flight Center's (GSFC) Information Systems Center.

9. REFERENCES

1. K. Wolf, et al, "Expert System Technology in Observing Tools", *Proceedings of SPIE Vol. 4010*, 2000.
2. A. Koratkar, et al, "NGST's Scientist's Expert Assistant: Evaluation Plans and Results", *Proceedings of SPIE Vol. 4010*, 2000.
3. J. Jones, et al, "Next Generation User Support Tools", *Journal of the Brazilian Society of Mechanical Sciences, Vol. XXI*, pp. 84-89, 1999.
4. A. Koratkar and S. Grosvenor, "The Next Generation User Support Tools", *Astronomical Society of the Pacific Conference Series, Volume 172*, pp. 57-64, 1998.
5. T. Brooks, et al, "An Expert Assistant System to Support the General Observer Program for NGST", *Proceedings of SPIE Vol. 3349*, pp. 450-455, 1998.
6. T. Brooks, et al, "Visualization Tools to Support Proposal Submission", *Proceedings of SPIE Vol. 3349*, pp. 441-449, 1998.
7. S. Grosvenor, et al, "Exploring AI Alternatives in Support of the General Observer Program for NGST", *Proceedings of the International Workshop on Planning and Scheduling for Space Exploration and Science*, 1997.